Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCSE-2014-53

2014

Inferring Memory Map Instructions

Paul T. Scheid, Ari J. Spilo, and Ron K. Cytron

We describe the problem of inferring a set of memory map instructions from a reference trace, with the goal of minimizing the number of such instructions as well as the number of unreferenced but mapped storage locations. We prove the related decision problem NP-complete. We then present and compare the results of two heuristic approaches on some actual traces.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Scheid, Paul T.; Spilo, Ari J.; and Cytron, Ron K., "Inferring Memory Map Instructions" Report Number: WUCSE-2014-53 (2014). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/110

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2014-53

Inferring Memory Map Instructions

Authors: Paul T. Scheid, Ari J. Spilo, and Ron K. Cytron

Abstract: We describe the problem of inferring a set of memory map instructions from a reference trace, with the goal of minimizing the number of such instructions as well as the number of unreferenced but mapped storage locations. We prove the related decision problem NP-complete. We then present and compare the results of two heuristic approaches on some actual traces.

Type of Report: MS Project Report

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

Inferring Memory Map Instructions*

Paul T. Scheid, Ari J. Spilo, and Ron K. Cytron
Washington University

August 6, 2014

Abstract

We describe the problem of inferring a set of *memory map* instructions from a reference trace, with the goal of minimizing the number of such instructions as well as the number of unreferenced but mapped storage locations. We prove the related decision problem NP-complete. We then present and compare the results of two heuristic approaches on some actual traces.

1 Introduction and Problem Statement

The problem we consider in this paper arose from the following situation. We are using two tools, `valgrind` and `cachesim`. The `valgrind` tool produces a trace of a program's load and store operations as show below:

```
S:100:8
L:112:8
S:400:8
L:412:8
```

The first character specifies whether the reference is a store (S) or load (L) instruction. The next field specifies the starting address of the affected storage. The last field specifies the length of the affected storage, in bytes. The above sequence corresponds to the shaded regions shown in Figure 1.

This trace is then supplied to the `cachesim` tool, which allows experimentation with new cache-management algorithms and provides statistics about the algorithms' effectiveness. That tool requires not only the trace but also a specification of the regions of storage that the traces references will access. These regions are usually declared by a running program using operating system calls such as `sbrk` to establish or increase the data segment of the program. The `valgrind` traces unfortunately do not contain any direct capture of instructions that

*This work was supported by the National Science Foundation through grant EAGER 1237425.

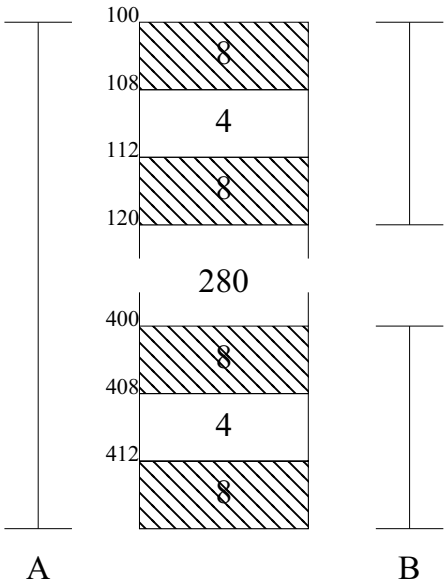

Figure 1: Example of mapped and unmapped memory regions. Each region is labeled with its starting address. The shaded regions, each of size 8 bytes, are referenced by the program. The unshaded regions, of sizes 4 and 280 bytes, are never referenced by the program. Two possible interval constructions are shown, A and B, which result in densities of .1 and .8, respectively.

establish data segment regions. The problem we therefore address here concerns inferring such regions from the references in a trace.

Here, we call each such specified region an *interval*. Given the storage references of a program, one such specification could simply be a *single* interval sufficiently large to capture all of the references. Interval A in Figure 1 is such a specification for the trace shown above. We define the *density* of an interval as the ratio of the interval's referenced bytes to its total number of bytes. The density of Interval A is $\frac{32}{320}$ or 0.1. If this single (large and low-density) interval is supplied to `cachesim`, then that tool would allocate a single 420-byte region of storage, but most of those bytes would never be referenced by the trace. While the example shown in Figure 1 wastes only 288 bytes, most real programs would waste much more space with a single such declared region. The relatively large gap in the middle of Figure 1 typically separates the stack and heap areas of a program, so that they can grow toward each other without interference. In a 64-bit address space, the middle gap would be much larger, and a single interval could consist of the entire 64-bit address space.

On the other hand, consider the intervals shown as B in Figure 1. This specification results in two intervals, each of density $\frac{16}{20}$ or 0.8. Although not shown in Figure 1, intervals can always be found with unity density using the storage-referencing instructions themselves. In our example, four such intervals would result from this treatment, each containing 8 bytes. However, in a typical address trace, the result of such a construction would be millions of intervals. Such as specification, while correct, would require excessive time to process by

`cachesim`.

Given a trace of a program's the load and store operations, we seek to infer the fewest regions, subject to a minimum required density for each such region. The rest of this paper is organized as follows. Section 2 presents a decision problem and proves that it is NP-complete. Section 3 presents two heuristics and compares their speed and effectiveness.

2 Related Decision Problem

To show the NP-hardness of our problem we restate it as a decision problem:

> *Given a trace of a program's load and store instructions, can we formulate a set of intervals that wastes exactly K bytes?*

In other words, the inetrvals found would contain all of the storage locations referenced by the trace, but they would also contain exactly K unreferenced locations.

To prove the above problem NP-complete, we offer a reduction from the subset sum problem:

> *Given as set of integers $S = \{n_1, n_2, \ldots, n_T\}$, and an integer K, is there a subset of S whose values sum exactly to K?*

We use the notation $[\,a, b\,)$ to denote the half-closed, half-open interval that starts at a and runs up to, but not including, b. Such an interval contains $b - a$ bytes. From an instance of subset sum, we create the following instance of our decision problem:

Number	Instruction	
1	S: 0	:1
	[Gap n_1]	
2	S: $n_1 + 1$	:1
	[Gap $K + 1$]	
3	S: $n_1 + 1 + (K + 2)$	:1
	[Gap n_2]	
4	S: $n_1 + 1 + (K + 2) + n_2 + 1$	:1
	[Gap $K + 1$]	
5	S: $n_1 + 1 + (K + 2) + n_2 + 1 + (K + 2)$	:1
.	. .	.
	[Gap $K + 1$]	
$2T$-1	S: $\sum_{i=1}^{T-1} n_i + (T - 1) + (T - 1)(K + 2)$	:1
	[Gap n_T]	
$2T$	S: $\sum_{i=1}^{T} n_i + (T) + (T - 1)(K + 2)$	:1

These instructions are created so that there is a gap for each integer in the set S. Between each pair of referenced locations there is a gap that corresponds either to an integer from the set S or that is exactly $K + 1$ bytes. Thus, if there exists an interval construction for

the above set of references that wastes exactly K bytes, there must be a subset of S whose values sum to K, and *vice versa*.

As an example, consider the set $S = \{1, 5, 7, 12\}$ with $K = 13$. The resulting instructions are shown in Figure 2(a), and a solution exists by creating an interval for references 1 and 2, and for 7 and 8. These intervals waste $1 + 12 = 13$ bytes. The other references are each in their own interval, wasting no space. On the other hand, no intervals can be merged in Figure 2(b) such that the number of wasted bytes is 16.

Number	Instruction
1	S: 0 :1
	[Gap 1]
2	S: 2 :1
	[Gap 13 + 1]
3	S: 17 :1
	[Gap 5]
4	S: 23 :1
	[Gap 13 + 1]
5	S: 38 :1
	[Gap 7]
6	S: 46 :1
	[Gap 13 + 1]
7	S: 61 :1
	[Gap 12]
8	S: 74 :1

(a)

Number	Instruction
1	S: 0 :1
	[Gap 1]
2	S: 2 :1
	[Gap 16 + 1]
3	S: 20 :1
	[Gap 5]
4	S: 26 :1
	[Gap 16 + 1]
5	S: 44 :1
	[Gap 7]
6	S: 52 :1
	[Gap 16 + 1]
7	S: 70 :1
	[Gap 12]
8	S: 83 :1

(b)

Figure 2: Construction applied to $S = \{1, 5, 7, 12\}$ with (a) $K = 13$ and (b) $K = 16$. For (a), a solution exists if intervals are created for the first two instructions and last two instructions. For (b), no solution can be found.

Thus, any instance of subset sum can be translated (in polynomial time) to an instance of our decision problem. A solution to our decision problem can be verified in polynomial time. Thus, our decision problem is NP-complete, proving our original problem NP-hard.

3 Two Heuristic Approaches

With the optimization proven NP-hard, we next turn to two heuristics for determining intervals from address traces. We begin with a set of intervals, one for each instruction in the original trace. Such a collection of intervals has perfect (unity) density, but we seek a much smaller set of intervals.

The original set of intervals is made smaller by repeatedly *merging* pairs of intervals still in the set. The heuristics we present differ in the way they consider candidates for merging.

For each interval I in the set, the "all pairs" heuristic considers the density that would result from merging I with each other interval in the set. If a merge with J provides the highest density, and if that density is above our required threshold, then intervals I and J are replaced by a single interval K that merges them. Once K enters the set, it also is considered for merging with each other interval in the set.

4 Experiments

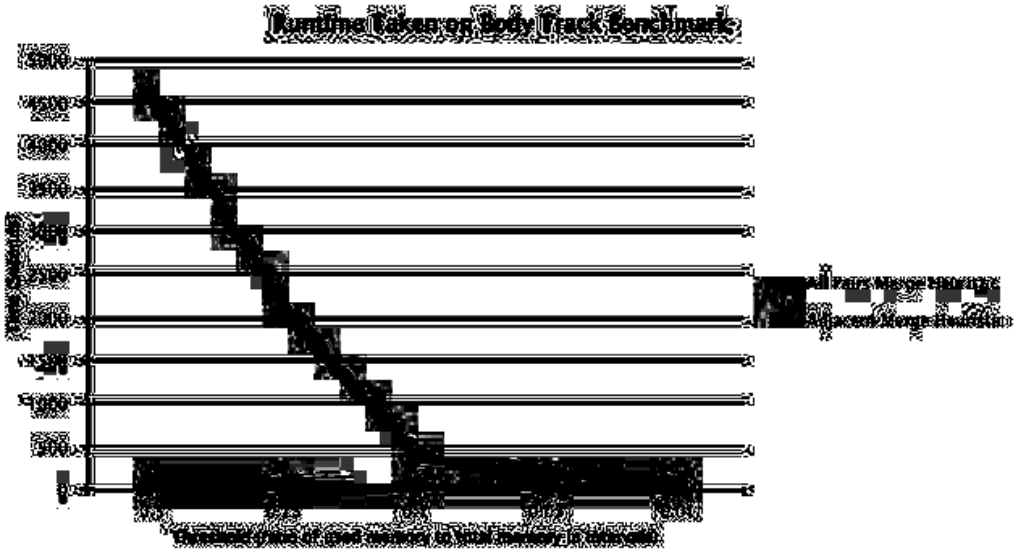

Figure 3: Comparing the runtime of our two heuristics. We generally sought density thresholds of 0.1. For this benchmark, the faster heuristic computed its results in seconds, compared to ∼7 minutes for the all pairs heuristic.

We tested these heuristics on multiple benchmark programs from the `parsec` benchmark suite [1]. Given a certain threshold, we prefer a result that has fewer intervals over one that has more. We found that the all-pairs heuristic and the adjacent-pairs heuristic generate interval quantities within approximately 1% of each other. While the all-pairs merge heuristic merges intervals that result in the highest density, based on our data, we conclude that the best possible merge is also generally found in the adjacent intervals, explaining why the number of intervals generated by the two heuristics is always so similar.

Although they generate similar quantities of intervals, the adjacent merge heuristic is significantly more efficient in terms of the amount of time taken to generate the intervals. This is because the all-pairs merge heuristic considers all intervals when attempting to merge, taking $O(n^2)$ time, and the adjacent merge heuristic only considers its neighbors in the ordered set, taking $O(n \log n)$ time. The runtimes of both heuristics are shown in Figure 3.

5 Conclusion

While no approach can hope to solve the problem exactly due to its NP-complete nature, our adjacent merge approach generates a fairly small number of intervals compared to the files size while still taking a reasonable amount of time. We believe that this approach is more practical than the all-pairs approach because there is no evidence that checking all merge candidates results in fewer intervals and that approach takes significantly more time. Therefore, we believe that in this scenario, the adjacent merge approach is a better solution to the subset sum problem than the all-pairs approach.

References

[1] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.